

useMemo Hook:

What's useMemo hook:

It's a hook which is used in **functional component** and return the **memoised value**, where we are **caching the value**, so that it doesn't need to be **recalculate it**, it's **based on the dependency change**, and it will help **to improve the performance**.

When to Use useMemo:

- When you deal with **heavy computations**.
- When the **same computation is repeated multiple times** and **returns the same result for the same inputs**.
- When **rendering large lists or data grids** and you want to avoid unnecessary re-renders.

When Not to Use React useMemo:

We should not use useMemo **for memoizing a function such as a callback**.

How is it work:

`const memoizedResult = useMemo(function, dependencies);`

- During **initial rendering**, `useMemo(function, dependencies)` invokes function, **memoizes the calculation result**, and returns it to the component.
- If the **dependencies don't change** during the next renderings, then `useMemo()` doesn't invoke function, but **returns the memoized value**. But if the **dependencies change** during re-rendering, then `useMemo()` invokes function, **memoizes the new value, and returns it**.
- If you pass an **empty array** (`[]`) as the dependencies, the memoized value will be computed only once during the initial render and will remain the same for subsequent renders.

Difference between useMemo vs useEffect:

It's important to note that we shouldn't add any code to use `useMemo` that we **don't want to be run when the page or component is being rendered**. **Any code that affects another component than the current one** (called side effects) **should be kept in a `useEffect`**.

Difference between useMemo vs useCallback:

The `useCallback` hook allows you **to memoize the entire function**, and the `useMemo` hook allows you to **memoize the output of functions**. Also, let's have the difference between while using it.

```
import { useCallback } from 'react';

function MyComponent({ prop }) {
  const callback = () => {
    return 'Result';
  };
  const memoizedCallback = useCallback(callback, [prop]);

  return <ChildComponent callback={memoizedCallback} />;
}
```

```
import { useMemo } from 'react';

function MyComponent({ prop }) {
  const callback = () => {
    return 'Result';
  };
  const memoizedCallback = useMemo(() => callback, [prop]);

  return <ChildComponent callback={memoizedCallback} />;
}
```

Example:

Suppose you have an increment and decrement counter. Also, you have a multiply function. Now when you call the multiply function where we are multiplying with the increment (i.e, no usage of decrement). When you click the increment button the multiply function will get call which is as expected because we are using the increment, but when you click on decrement button then there is a multiply function ALSO, getting call.WHY ?

The screenshot shows a VS Code editor with a file named `App.js`. The code defines a React component `App` with two state variables: `increment` (initial 0) and `decrement` (initial 100). A `multiply` function is defined as `const multiply = () => { console.log("multiply called"); return increment * 100; }`. The component renders two buttons: "Increment" and "Decrement". The "Increment" button calls `setIncrement(increment + 1)`, and the "Decrement" button calls `setDecrement(decrement - 1)`. The state is displayed in the UI: "You clicked {increment} increment times, Multiply: {multiply()}" and "You clicked {decrement} decrement times". The console shows multiple "multiply called" messages, indicating that the `multiply` function is being called multiple times, even when the decrement button is clicked. This is because the `multiply` function is not memoized and its return value is recalculated on every render.

You can restrict the Multiply function to get call on decrement button using the useMemo hook, where you need to pass the dependency array and based on the dependency it will call.

The screenshot shows a VS Code editor with a file named `App.js`. The code is similar to the previous example, but the `multiply` function is now defined using `useMemo`: `const multiply = useMemo(() => { console.log("Multiply called"); return increment * 100; }, [increment]);`. The component renders two buttons: "Increment" and "Decrement". The state is displayed in the UI: "You clicked {increment} increment times, Multiply: {multiply}" and "You clicked {decrement} decrement times". The console shows only one "Multiply called" message, indicating that the `multiply` function is only called when the `increment` state changes, and not when the `decrement` state changes. This is because the `multiply` function is memoized and its return value is only recalculated when the dependency array `[increment]` changes.

What's the difference between useMemo and React.memo in reactjs:

Aspect	useMemo	React.memo
Purpose	Memoizes the result of a function call based on dependencies	Memoizes the rendering of a functional component based on its props
Usage	Used inside functional components to optimize expensive calculations or data transformations	Used to optimize functional components by preventing unnecessary re-renders based on prop changes
Input	Function and dependencies	Functional component
Returns	Memoized value	Memoized component
Dependencies	Re-runs the function when dependencies change	Re-renders the component when props change
Performance	Optimizes calculations within the component	Optimizes rendering of the component
Example Use Case	Memoizing a complex calculation within a component	Memoizing the rendering of a component with static props